Cahier des Charges

PFA

Projet ICECAST

Tuteur pédagogique : Bertrand Le Saëc

Tuteur scientifique : Pierre Jarillon

Antoine Allombert
Arnaud Ebalard
Mickaël Floc'hlay
Sébastien Henrio
Guillaume Le Roux
Florian Pierron
Denis Tessier
Jérémy Vies

Table des matières

Chapitre 1

Introduction

Notre projet de fin d'année a pour but de poursuivre le développement d'**ICECAST**. Ce programme est un serveur permettant de diffuser des sources audio en streaming, aux formats **MP3** ou **Ogg Vorbis**. Le format de compression audio proposé par Vorbis utilisant un algorithme libre, ICECAST a pour but de proposer une alternative libre aux solutions commerciales existantes.

Notre rôle dans ce projet sera d'améliorer la solution existante, nous serons par conséquent amenés à modifier les sources d'ICECAST. D'autre part, nous devrons fournir la documentation nécessaire pour l'installation, l'administration et la maintenance du programme. Enfin, nous devrons améliorer le processus d'installation.

1.1 L'environnement du projet

Ce projet est réalisé en collaboration avec une quinzaine de développeurs à travers le monde. Il s'appuie sur un existant déjà conséquent, mais que nous devons néanmoins remanier. La synchronisation entre les différents collaborateurs se fait grâce à un **CVS** placé sur **Savannah** (savannah.nongnu.org). Le programme est codé en **C**, et la compilation se fait grâce aux outils **autoconf** et **automake**.

1.2 Contenu de ce cahier des charges

Dans ce cahier des charges nous traiterons tout d'abord de l'état actuel des sources. Nous exprimerons ensuite les besoins fonctionnels et non fonctionnels.

Chapitre 2

Présentation de l'existant

Ce chapitre présente rapidement les différents modules en décrivant brièvement ce qu'ils font. Il s'agit en quelque sorte d'une revue du code de l'application. Ceci permet de donner une idée de l'existant sur le projet et de présenter les limites et problèmes que nous avons pu déceler. Cette présentation n'a pas pour but d'être exhaustive mais plutôt de fournir une vision de notre point de départ dans le but de justifier de futurs développements.

2.1 Présentation globale des modules d'ICECAST

Ici, nous présentons les principaux modules utilisés lors du fonctionnement d'ICECAST. Les détails sur le contenu de ces modules sont donnés dans la section suivante.

ICECAST est une application fonctionnant sous la forme d'un unique processus multithread. Elle a pour rôle de servir en streaming audio des clients distants qui viennent se connecter.

Pour cela, elle dispose d'un premier module `connection` qui gère l'arrivée des clients et l'aiguillage sur le module de traitement adéquat pour traiter leurs requêtes et un module de gestion des sockets `sock`. Les requêtes des clients sont reçues en HTTP et sont parsées par le module `httpp`.

Si le client fait la demande d'un fichier local non streamé, c'est le module `fserve` qui est appelé. S'il s'agit d'une demande d'un fichier streamé distant (serveur distant ou streamer local de type **ICES**), les modules `source` et `slave` sont utilisés.

Il est à noter qu'ICECAST est capable de fournir au client des fichiers audio streamés et non streamés de format MP3 ou Ogg Vorbis. Trois modules fournissent l'interface pour ces différents services : `format`, `format_MP3` et `format_vorbis`.

La configuration d'ICECAST se fait par l'intermédiaire d'un fichier **XML** fourni au démarrage du serveur. Pour traiter celui-ci, ICECAST requiert la présence sur le système des librairies `libxml2` et `libxslt` et les utilisent par l'intermédiaire du module `xslt`.

Le serveur est capable de gérer le log des nombreux évènements qui peuvent se produire (connexion, ajout d'une source, ...) par l'intermédiaire des modules

`log` et `logging`.

Pour finir, l'application étant multithread, il existe un module `thread` qui réalise l'abstraction de la gestion des threads en fonction du système sur lequel ICECAST est installé.

2.2 Schéma de dépendance

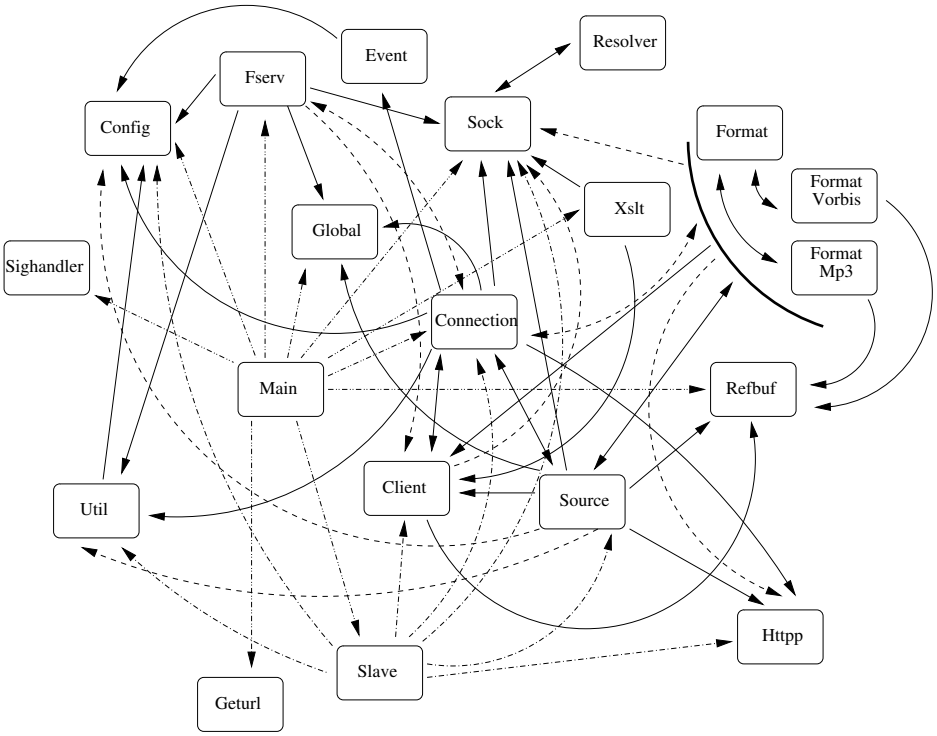

FIG. 2.1 – Graphe de dépendance des modules

2.3 Détails des modules

Dans cette section, nous présentons le début de notre première approche de l'existant avec un peu plus de détails. Chacun des différents modules que nous avons pour le moment étudié y est décrit : son utilité, ses problèmes et ses limitations y sont mentionnés.

Le `main`

Le `main` possède une structure assez simple à suivre : il lance l'initialisation des sous-systèmes puis la création du socket bindé au port d'écoute d'ICECAST.

Remarques : Les utilisations des structures des autres modules ne sont pas toujours très propres (manipulation directe de données sans utilisation d'interface). Il faut modifier cela pour améliorer la maintenabilité du programme.

Module `client`

Ce module a pour rôle de définir une structure associée à tout client. Il fournit les fonctions permettant de créer et de détruire ce type de structure. De plus, il fournit 3 fonctions pour envoyer des messages d'erreurs au client.

Remarques : Ces fonctions d'envoi de message au client ne devraient pas se trouver ici. De plus, ce module devrait fournir des fonctions d'accès et de modifications des structures : dans le reste du code, tout se passe par accès direct au structures de données. Il serait possible de gagner en modularité et en maintenabilité.

Module `connection`

Ce module définit la structure de connexion et fournit des fonctions pour tester les requêtes des clients et les diriger en fonctions du contenu de celles-ci (mots de passe, demandes, ...). On y trouve aussi des fonctions internes de gestion de listes de threads et d'envoi de signaux de réveil des threads endormis.

Remarques : Les fonctions internes de gestion de listes de threads ne devraient pas se trouver là mais plutôt dans le module de threads ou au moins dans un module séparé. Ici encore, les accès directs aux structures des autres modules sont nombreux. Il devrait y avoir plus d'abstraction.

Module `thread`

Ce module fournit les structures permettant de gérer les threads et notamment les sémaphores, verrous et autres variables de condition.

Remarques : Les fonctions de ce module sont des encapsulations des fonctions du module système de gestion des threads. On réalise ainsi une abstraction de plus haut niveau permettant de s'éloigner du type de système sur lequel tourne le serveur. Au début de ce fichier, on trouve de nombreuses macro-fonctions concernant les logs. Elles ne devraient pas se trouver ici. De plus, dans ce module on a des chaînes de redéfinitions de fonctions qui rendent la maintenance difficile.

Module `fserve`

Ce module s'occupe des clients qui demande à accéder à des fichiers stockés localement et non streamés par un ICES. Il y a ainsi un thread qui tourne en permanence pour répondre à ces requêtes. Les clients que l'on doit servir sont stockés dans des structures d'arbres.

Remarques : Le stockage des clients dans un arbre est coûteux et ce type de structure n'apporte pas ici de véritable intérêt. En effet, la structure d'arbre induit une hiérarchie des clients, ce qui n'a aucune utilité ou légitimité. Il faudrait donc préférer à ce strockage en arbre une manière de stockage séquentiel des clients (par une liste par exemple).

Module `slave`

Le module slave a pour but de gérer la réception de données pour des sources (audio) distantes. On a ainsi un thread qui tourne en permanence pour vérifier périodiquement s'il y a des nouvelles sources sur le serveur maître. On a aussi

une fonction qui permet d'initialiser la connection entre une source et notre serveur ICECAST afin de préparer la réception du fichier streamé.

Module `sighandler`

La gestion des signaux est centralisée dans le module `sighandler`. Lorsque le serveur reçoit un `sigkill`, on l'arrête proprement. La reception d'un signal `sighup` est prévue mais non encore implémentée. Un tel signal servira à mettre à jour l'état de la configuration.

Module `avl`

Ce module permet de gérer des arbres génériques binaires de recherche équilibrés. On a des fonctions de création, de suppression, de recherche et d'accès.

Remarques : Les fonctions définies dans cette bibliothèque sont très souvent utilisées par ICECAST. De nombreuses utilisations de ces arbres nous paraissent abusives.

Module `sock`

Ce module définit les fonctions relatives aux sockets : leur création, l'écriture, de lecture, la connection, l'écoute et les attentes. Il est utilisé par la plupart des modules principaux qui réalisent des connexions et des transferts de données.

Module `resolver`

Ce module sert à déterminer un nom de machine à partir de l'adresse IP, et inversement, ce qui est utile pour le module `sock`.

Remarques : Les 2 modules `sock` et `resolver` font appel l'un à l'autre, et leur initialisation sont quasi-identiques. Peut-être ceux-ci pourraient mieux coordonnés, voire être modifiés.

Module `format`

Ce module définit une structure générique sur les formats de données. Cette structure contient les différentes fonctions permettant de communiquer les données aux clients. A savoir, les headers, les métadonnées et les données à proprement parler. Cette structure contient également des fonctions permettant de récupérer les données depuis la source. En plus de la structure, sont définies deux fonctions, l'une créant la structure précédemment définie, l'autre écrivant le contenu d'un buffer sur une socket, et deux fonctions de reconnaissance du type de fichier. Ce module est en relation directe avec les modules `format_mp3` et `format_vorbis` dans lesquels sont définies les structures et fonctions spécifiques à ces types de fichiers.

Remarques : L'organisation des trois modules `format`, `format_mp3` et `format_vorbis` est un peu confuse dans la mesure où certaines définitions paraissent redondantes, notamment les deux fonctions définies dans le module `format` qui possèdent des équivalents dans les deux autres modules.

Module `format_mp3`

Ce modules contient les fonctions spécifiques à l'envoi des données pour un fichier de type MP3. Les fonctions présentes dans ce module permettent :

- la récupération de métadonnées,
- le formatage des métadonnées,
- l'envoi des métadonnées,
- l'envoi des headers,
- l'écriture des données via une chaîne de buffers,
- l'écriture de "prédonnées" (informations envoyées avant les données audio),
- la création d'un `plugin_mp3` contenant des pointeurs vers les fonctions définies dans le module.

Remarques : Ce module comporte un certain nombre d'erreurs. De plus la différence entre les prédonnées et les headers paraît obscure. Les informations qu'ils contiennent nous échappent également. Nous pensons trouver ces réponses sur les normes de formatage dans les sources de `libshout` et ICES. En outre, il faudra peut-être modifier ces normes et donc les fonctions de ce module si nous devions ajouter des informations dans le flux.

Module `format_vorbis`

Ce module est l'équivalent du précédent pour le format Vorbis.

Module `stats`

Ce module permet de gérer et d'accéder à des statistiques sur les connexions au serveur.

Remarques : L'opacité de ce module ne nous permet pas pour le moment de le décrire en détail.

Module `source`

Ce module gère l'envoi de liaisons actives entre une source distante et l'ensemble des clients connectés à cette source.

Module `refbuf`

Ce module gère une liste chaînée de buffers. Ces buffers servent à stocker les données à transmettre à chaque client. Ils sont remplis par les modules `format` et vidés par `source`.

Remarques : Il n'y a aucune gestion d'erreur dans ce module.

Module `xslt`

Ce module gère les feuilles de conversion des données XML vers un autre format. Ces feuilles de conversion sont stockées dans un cache géré par ce même module.

Module `yp`

Ce module gère les transmissions vers les serveurs de gestion des bases de données des sources disponibles. Il met en forme les données de chaque source dans un url envoyé au serveur défini par la configuration.

Module `httpp`

Ce module sert à parser une requête HTTP. La structure de données associée permet de stocker la méthode de la requête parsée, l'uri visée et, dans 2 arbres (`avl_tree`), les variables et leurs valeurs précisées dans la requête http ou dans l'uri. Les nom de fonctions sont suffisamment clairs. Elles permettent le parsing, la création, l'initialisation et la libération de la structure ainsi que l'accès aux variables.

Remarques : Quelques aspects de ce module sont gênants.

- L'arbre nommé `queryvars` n'est apparement utilisée par aucun module.
- Des méthodes de requêtes sont décrites mais non utilisées (toutes sauf `get`).
- Des portions de code paraissent assez douteuses (de nombreuses optimisations sont possibles).
- Certaines fonctions (du `.c`) sont assez obscures : notamment une fonction `(url_escape)` qui transforme un hexadécimal en décimal (exprimé en chaîne de caractères).
- La fonction `parse_query` est clairement annoncée comme incorrecte.

Module `config`

Ce module présente la structure de données utilisée pour stocker les informations fournies par le fichier XML ainsi que les fonctions qui la manipulent : création, initialisation, mise à jour des différents champs, destruction.

Remarques :

- Deux fonctions restent à écrire en totalité.
- Deux fonctions ne sont pas terminées (les points manquants concernant la compatibilité avec les anciennes versions et un bug persiste au niveau des mots de passe).
- Le code me paraît simplifiable (notamment avec l'emploi de *switch* à la place de *else if* à répétition).
- En revanche, la gestion d'erreur est correcte (dans le sens où nous n'avons pas vu de possibilité d'erreur non-traitée).

Module `log`

Ce module permet de gérer l'ensemble des fichiers logs. Pour cela, un tableau de structures log est créé en variable globale. Une structure log correspond à un fichier spécifique sur le système de fichiers. Ce module permet d'initialiser l'ensemble des fichiers logs, de les ouvrir, d'écrire des données et de les fermer.

Remarques : La gestion des fichiers logs ne se fait pas d'un point de vue global, dans le sens où tous les modules qui désirent écrire dans un fichier de log le font par un accès direct au fichier. Donc, les choix de l'utilisateur sur les données à écrire dans les fichiers de log sont testés séparément dans chaque module qui désire loguer des informations et non pas centralisés par un daemon

de log (par exemple) qui pourrait avoir les critères de sélections correspondant aux choix de l'utilisateur.

Module `util`

Ce module permet de récupérer l'header d'une requête http, de vérifier que l'uri est bien formée : il n'accepte pas les chemins de la forme `foo/../` afin d'éviter les attaques sur la racine du système de fichiers.

Remarques : Aucun retour des fonctions d'allocation mémoire (`malloc` et `calloc`) n'est testé. Il n'y a donc pas suffisamment de gestion d'erreurs.

Module `global`

Ce module définit la structure globale de ICECAST : le nombre de clients, le nombre de sources, l'état et le socket du serveur, l'arbre des sources (les sources sont les fichiers audio streamés). Le module permet d'initialiser les champs suscités et de positionner un verrou sur cette structure.

Module `geturl`

Ce module reprend les fonctionnalités de la bibliothèque *libcurl*.

2.4 Découpage par activité

2.4.1 Schéma global par activité

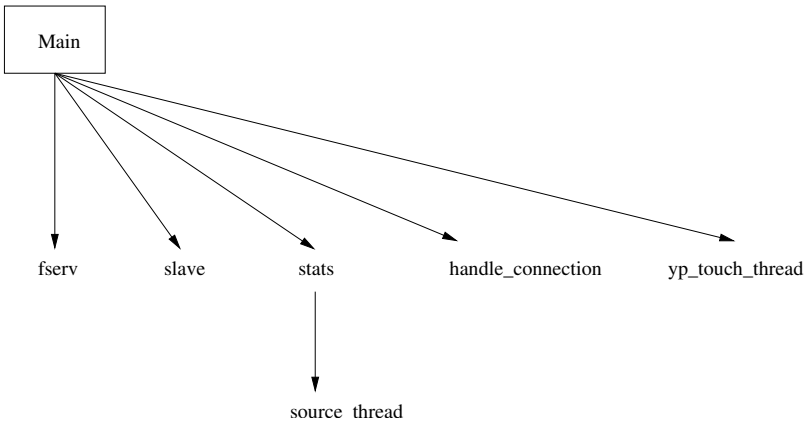

FIG. 2.2 – Schéma global par activité

2.4.2 `fserve` : serveur de fichiers locaux

Un thread joue le rôle de serveur de fichiers locaux. Il sert les clients qui ont fait la requête d'un fichier *non streamé*, c'est à dire local pour le serveur. Pour cela, 2 structures d'arbres stockent les clients :

- Le `pending_tree`, qui se charge du stockage des clients qui viennent d'arriver et qui n'ont pas encore été servis.
- Le `client_tree`, qui se charge du stockage des clients qui sont en train d'être servis.

Le fonctionnement du thread se décompose ainsi :

- Si les 2 structures sont vides, on attend qu'un client arrive.
- On sert les clients qui sont dans le `client_tree`, en essayant de leur envoyer des données audio. On repère les clients que l'on a fini de servir (fin de leur morceau de musique).
- On fait un deuxième passage dans le `client_tree` pour supprimer les clients que l'on a fini de servir.
- On fait un parcours du `pending_tree` pour y prendre les clients et les insérer dans le `client_tree`.
- On reboucle (ainsi, les clients qui viennent de passer du `pending_tree` au `client_tree` vont être servis).

2.4.3 `stats` : statistiques sur les évènements

Un thread se charge de maintenir des statistiques sur les évènements qui se produisent (nombre de connexions actuelles, nombre de personnes écoutant une source, ...). Pour cela, il dispose d'une liste d'évènements et de 2 structures d'arbres :

- `_global_event_queue` est une liste dans laquelle sont placés les évènements qui sont à traîter au niveau des statistiques.
- `_stats.source_tree` se charge du stockage des statistiques concernant les sources streamées.
- `_stats.global_tree` se charge du stockage des statistiques générales.

Pour maintenir les statistisques à jour, les rapports des différents événements qui ont été placés dans la liste `_global_event_queue` sont traités et en fonction de leur nature, ils sont placés dans le `_stats.source_tree` ou `_stats.global_tree`. Ainsi, toutes ces statistiques sont accessibles rapidement par accès aux arbres.

2.4.4 `slave` : récupération des nouvelles streams d'un serveur maître et lancement des threads de traitement

Un thread se charge de récupérer sur son serveur maître défini dans la configuration la liste des *fichiers streamés* que celui-ci diffuse. Pour chaque stream que le serveur local ne diffuse pas encore (stream ajoutée sur le serveur maître depuis la dernière vérification), il y a création d'un nouveau thread `source_thread` qui va se charger de servir les clients qui font la requête d'écoute de cette source. Pour cela, chaque thread associé à une source distante streamée se charge de récupérer en temps réel les données de streaming envoyées par le serveur maître et travaille en parallèle d'une manière similaire à celle du thread `fserve`. En effet, chaque thread possède une structure de `pending_tree` et une autre de `client_tree`.

2.4.5 `handle_connection` : redirection des clients après lecture de leur requête

Un pool de threads se charge de gérer les demandes des clients en lisant leur requête puis en les redirigeant vers le module adéquat pour les servir. Pour cela, chacun des threads de ce pool peut accéder à une liste partagée des clients qui ont été acceptés. Les différentes redirections possibles sont les suivantes :

- `_handle_source_request` réalise les montages de sources streamées.
- `_handle_stats_request` envoie au client les statistiques en temps réel.
- `_handle_get_request` gère les demandes de fichiers des clients.

2.4.6 `yp_touch_thread` : envoie à un serveur de référencement (Yellow Pages) des noms de fichiers streamés

Pour chaque site de référencement défini et pour chaque source streamée que propose le serveur ICECAST, un thread est créé qui se charge d'envoyer les données au serveur de référencement (adresse du serveur ICECAST, port, bitrate, nom du programme, ...) à intervalle régulier.

Chapitre 3

Besoins fonctionnels

Afin de combler les manques du logiciel actuel, il nous semble bon d'intégrer de nouvelles possibilités et politiques de gestion du programme.

3.1 Configuration à distance

Il est à signaler qu'aucun existant n'est présent au début du projet.

3.1.1 Politique d'administration

Choix possibles

L'alternative qui se présente à nous est la suivante : pour administrer le logiciel à distance, on peut choisir de créer une page php dans laquelle un formulaire sera rempli. La validation de ce formulaire créera un fichier XML sur le disque du serveur ICECAST qui pourra alors servir à la configuration d'une source. L'autre solution possible est de passer par une interface type **webmin** et de créer directement le fichier à distance.
L'intérêt de la première solution est la facilité de la maintenance (interface facilement modifiable). En revanche, le désavantage est l'obligation pour l'administrateur de posséder un serveur **Apache**. Bien sûr, ce qui est un avantage pour la première solution est un inconvénient pour la seconde...

- **Travail à faire :**
  - Choix d'une solution.
  - Implémentation de la solution.
- **Estimation du temps requis :** 4 semaines (du 14/03/03 au 11/04/03)
- **Equipe :** Antoine Allombert, Mickaël Floc'hlay et Guillaume Le Roux

3.1.2 Reconfiguration à chaud

Présentation

Un administrateur peut reconfigurer les paramètres du fichier XML initial en cours d'exécution. Il faudra donc veiller au respect de la prise en compte des nouveaux paramètres par le logiciel après avoir préalablement choisi une politique parmi les différents choix possibles.

- **Travail à faire :**
  - Sélection d'une politique de reconfiguration.
  - Ajout de la fonctionnalité de prise en compte d'un changement du fichier de configuration.
- **Estimation du temps requis :** 2 semaines (du 20/04/03 au 03/05/03)
- **Equipe :** Antoine Allombert, Mickaël Floc'hlay et Guillaume Le Roux

3.1.3 Documentation pour l'administration

Caractéristiques

Cette documentation s'adresse bien sûr, en priorité, aux administrateurs du logiciel. Il faudra donc actualiser régulièrement cette documentation et la tenir à jour dès l'implémentation de nouvelles fonctionnalités d'administration. Ce travail de documentation se fera donc en parallèle au travail d'implémentation. Dans un soucis d'internationalisation, cette documentation sera tenue en français et en anglais.

- **Travail à faire :**
  - Ecriture de la documentation en parallèle à l'implémentation des nouvelles fonctionnalités.
  - Traduction de la documentation en anglais.
- **Estimation du temps requis :** 4 semaines (du 14/03/03 au 11/04/03)
- **Equipe :** Antoine Allombert, Mickaël Floc'hlay et Guillaume Le Roux

3.1.4 Identification des sources et sécurisation

Distinction source-client

Pour l'instant, une source se connecte sur ICECAST par le même port que les clients et doit donc pouvoir s'identifier comme telle pour différencier le traitement des connexions. Pour s'identifier, la source envoie son mot de passe en clair sur le réseau. Il faudra peut-être modifier cette politique et s'interroger sur l'opportunité de connecter sources et clients sur des ports différents : un port sécurisé pour l'authentification et un autre non-sécurisé pour l'échange des données. Les données n'ont pas besoin d'être sécurisées car elles sont destinées à être diffusées à tout le monde. Seul l'expéditeur des données doit s'authentifier de manière sécurisée.

- **Travail à faire :**
  - Définir les choix possibles et en choisir un.
  - Implémentation de la solution choisie
- **Estimation du temps requis :** 2 semaines (du 20/04/03 au 03/05/03)
- **Equipe :** Arnaud Ebalard, Sébastien Henrio, Florian Pierron, Denis Tessier et Jérémy Vies

Distinction inter-source

Nous devrons aussi redéfinir la stratégie que l'on devra adopter pour la gestion du montage d'une source par un utilisateur distant. En effet, pour demander de monter une source, il faut se connecter avec un mot de passe commun à toutes les sources, mais on doit faire attention à ne pas donner trop de droit à cet utilisateur, car il serait complètement anormal qu'il puisse supprimer la source d'un

autre client. Pour résoudre ce problème, chaque source possède un mot de passe qui permet de distinguer entre eux les utilisateurs sources. Il faudra définir ainsi le droit de chaque utilisateur ainsi que la politique à adopter en cas de conflit sur les sources montées. On pourra aussi se demander s'il doit y avoir un super-utilisateur ayant la possibilité de modifier toutes les sources afin de résoudre les éventuels conflits.

- **Estimation du temps requis :** 2 semaines (du 20/04/03 au 03/05/03)
- **Equipe :** Arnaud Ebalard, Sébastien Henrio, Florian Pierron, Denis Tessier et Jérémy Vies

Documentation de la sécurité

Pour gérer correctement la partie sécurité de notre projet nous devrons nous documenter sur la gestion des sockets securisés (SSL) et plus généralement sur les connexions securisés.

- **Estimation du temps requis :** 2 semaines (du 20/04/03 au 03/05/03)
- **Equipe :** Arnaud Ebalard, Sébastien Henrio, Florian Pierron, Denis Tessier et Jérémy Vies

3.1.5 Plaquette client

Dans une optique plus commerciale, nous rédigerons un document destiné aux organismes susceptibles d'utiliser ICECAST. Celui-ci présentera succinctement les différentes fonctionnalités du programme ainsi que le matériel nécessaire à son installation.

Chapitre 4

Besoins non fonctionnels

4.1 Amélioration du code existant

4.1.1 Etude du code existant

Le code actuel présente de nombreuses anomalies :
- absence de commentaires.
- dépendances incohérentes.
- plusieurs fonctions non implémentées.
- un bon nombre d'erreurs de codage.

Notre but est de réduire ces anomalies, afin de rendre le code plus efficace.

- **Travail à faire :**
  - Comprendre le code existant
- **Temps requis :** à définir.
- **Equipe :** à définir.

4.1.2 Modularisation du code

- **Travail à faire :**
  - Modification des modules existants.
  - Spécification des interfaces entre modules.
  - Graphe de dépendances entre modules.
  - Implémentation des modules.
- **Temps requis :** à définir.
- **Equipe :** à définir.

4.1.3 Test de modules

- **Travail à faire :**
  - Tester chaque module séparément.
  - Tester le programme.
- **Temps requis :** à définir.
- **Equipe :** à définir.

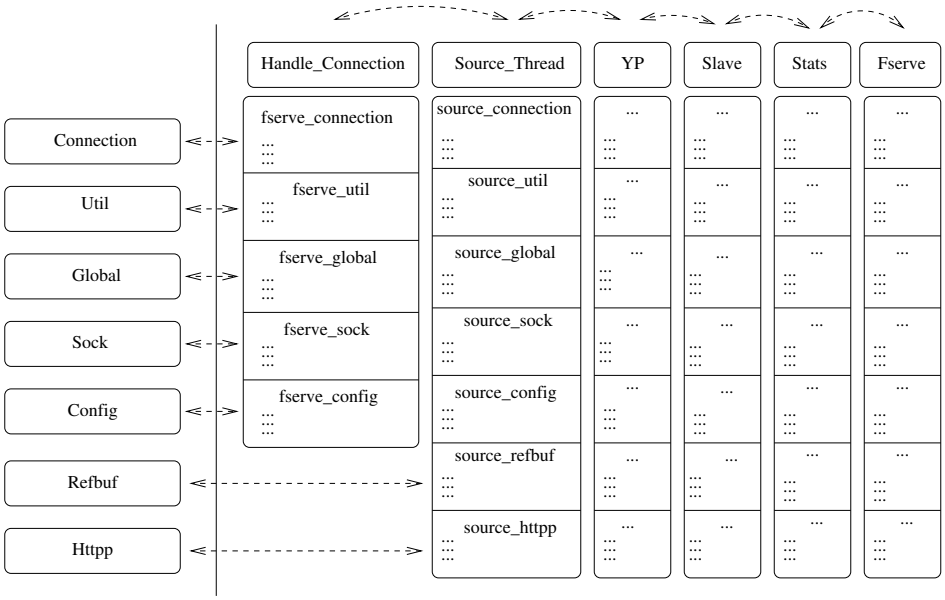

FIG. 4.1 – Schéma global par activité

4.1.4 Documentation de maintenance

Il s'agit de rédiger un manuel en anglais à l'attention des personnes désireuses d'avoir une connaissance précise de l'organisation des fichiers sources.

Seront présents dans cette documentation :

- une arborescence détaillée des dépendances entre les modules.
- une description générale de chaque module (les structures, les fonctions...).
- une explication précise des fonctions complexes.
- les limitations de fonctionnement du logiciel.
- les évolutions possibles.

Cela permettra une évolution du programme, diverses adaptations ou encore d'éventuelles corrections.

- **Travail à faire :**
  - Spécification des interfaces entre modules.
  - Graphe de dépendances.
  - Commentaires dans le code des modules.
  - Documentation anglaise de maintenance du code.
- **Temps requis :** à définir.
- **Equipe :** à définir.

4.2 Installation de Icecast

Notre objectif principal en ce qui concerne l'installation est de pouvoir compiler ICECAST et l'exécuter sur les principales plateformes **Linux** (Mandrake, Debian, Red Hat) et sur la plupart des **Unix** (Solaris, OpenBSD).

Nous allons essayer de faire une installation la plus simple possible pour un utilisateur non expérimenté, mais qui restera paramétrable pour les utilisateurs expérimentés.

L'installation devra gérer les problèmes de configuration et de compilation. Au niveau de la configuration, les bibliothèques indispensables à la compilation d'ICECAST devront être identifiées et installées si possible. Sinon, le programme d'installation devra afficher un message d'erreur clair permettant de donner l'origine du problème afin de faciliter la résolution du problème pour l'utilisateur.
Au niveau de la compilation, si la configuration s'est bien déroulée, il ne devrait pas y avoir de problèmes graves empêchant sa réalisation.

4.2.1 Réalisation des fichiers d'installation

Nous prévoyons de fournir des fichiers d'installation aux formats : `.rpm` (pour les **Red Hat** et les **Mandrake**), des `.deb` (pour les **Debian**) et des `.pkg` (pour **Solaris**). De plus, pour l'installation, la configuration et la compilation, seront utilisés les outils *automake* et *autoconf*.

- **Travail de l'équipe :**
  - Apprentissage des outils et formats d'installation.
  - Création des fichiers d'installation.
- **Temps requis :** à définir.
- **Equipe :** à définir.

4.2.2 Documentation d'installation

Nous nous engageons à fournir une documentation sur l'installation en anglais et en français afin que le logiciel ICECAST soit aisément installable par tous les utilisateurs.

- **Temps requis :** à définir.
- **Equipe :** à définir.

Chapitre 5

Glossaire

Autoconf : autoconf est un outil destiné à produire des scripts shell qui configurent automatiquement les packages de code source logiciel pour les adapter à de nombreuses sortes de systèmes de type Unix. Il permet donc une portabilité accrue et rend l'installation d'un logiciel plus facile pour un néophyte.2

Automake : automake est un outil destiné à générer automatiquement des fichiers Makefile.in (à partir des fichiers Makefile.am) qui permet de compiler un code source.2

Binder : Binder un socket consiste à l'affecter à un point de communication (un port par exemple).2

Bitrate : Nombre de bits nécessaire pour coder une seconde de musique ou de vidéo.2

CVS : CVS (Concurrent Versions System) est un système permettant de contrôler les différentes versions d'un système de fichiers. L'utiliser permet notamment de pouvoir avoir accès à toutes les versions d'un fichier - son historique - et permet ainsi de pouvoir faire "marche arrière", ce qui est très utile dans le cadre d'un développement de projet. Travailler avec un CVS est également très utile lorsque plusieurs personnes sont appelées à travailler sur un même fichier puisqu'il aide à résoudre les problèmes de conflits d'accès.2

Cache : Procédé consitant à stocker temporairement les données les plus fréquemment utilisées.2

Démon (daemon) : Un démon est un programme tournant en permanence sur la machine hôte et attendant une requête de service pour y répondre.2

En-tête (headers) : Données préliminaires donnant des informations sur des données à suivre.2

HTTP : HTTP : HyperText Transfer Protocol. HTTP est un protocole de transfert de données.2

ICES : ICES est un programme servant à streamer des fichiers audio MP3 et Ogg Vorbis.2

Log : Journal permettant d'enregistrer les divers évènements liés au serveur (exemple : nouvelles connexions, etc).2

Métadonnées : Informations sur les données quant à leur origine, contenu, localisation, structure, règles d'agrégation et de transformation.2

PHP : Langage de programmation internet interprété par un serveur - à la différence du HTML interprété par l'utilisateur2

Parser : Le parsing est une opération visant à donner un sens à des données formatées. Par exemple, pour un fichier XML, le parsing va consister, entre autres, à repérer les différentes balises, afin d'être capable de récupérer leur contenu.2

SSL : SSL : Secured Socket Layer. Protocole permettant la transmission sécurisée d'informations. Il assure l'authentification et la confidentialité des données échangées.2

Sémaphore : Variable utilisée pour spécifier (et donc également pour empêcher les conflits d'accès) les états de ressources partagées (en lecture, en écriture. . .).2

Serveur : Un serveur est en programme qui attend la connexion d'autres processus - les clients - qui viennent lui demander des informations. Son rôle est donc à la fois de traiter les demandes de ces clients et de leur répondre.2

Serveur Apache : Serveur web permettant l'interprétation du php2

Socket : Les sockets forment un dispositif logiciel pour la communication interprocessus, aussi bien sur une même machine que sur un réseau. Les sockets peuvent donc permettre la communication client/serveur.2

Streaming : Le streaming est un procédé de transmission de données par flux. L'accès à ces données peut donc se faire sans télécharger de fichiers entiers. Grâce au streaming, on peut donc avoir accès, par exemple, à des sources audio ou vidéo live sans consommation d'espace disque.2

Thread : Un thread est un "sous-processus" d'un processus affecté à une tâche particulière. On parle aussi de "processus léger".2

URI : Nom générique englobant un ensemble de méthodes d'identification des ressources sous Internet. URI : Uniform Resource Identifier.2

Webmin : Interface graphique en HTML pour administrer un serveur à distance.2

XML : XML : eXtensible Mark-up Language. Le XML est un langage offrant des mécanismes de structuration de données.2